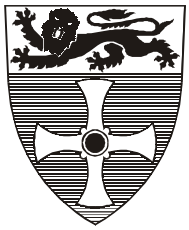
UNIVERSITY OF
NEWCASTLE



COMPUTING SCIENCE

A Co-operative Object-Oriented Architecture for Adaptive Systems

Rogério de Lemos

TECHNICAL REPORT SERIES

No. 674
July, 1999

Contact: Rogério de Lemos
r.delemos@newcastle.ac.uk
http://www.cs.ncl.ac.uk/people/r.delemos/

Published by the University of Newcastle upon Tyne,
Department of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK

A Co-operative Object-Oriented Architecture for Adaptive Systems

Rogério de Lemos
Department of Computing Science
University of Newcastle upon Tyne, NE1 7RU, UK
r.delemos@newcastle.ac.uk

Abstract

Adaptive systems should be able to adapt to changes that occur in their operating environment without any external human intervention. Software architectures for such systems should be flexible enough to allow components to change their pattern of collaboration depending on the environmental changes and goals of the system. A drawback for using object-oriented models for describing software architectures for adaptive systems is their lack of effective means to represent collaborative behaviour between objects, if we consider that the capability of a system to be adaptable depends on how objects, as rigid entities, co-operate. This paper describes a co-operative object-oriented architectural style for the development of software for adaptive systems. The applicability of the architectural style is demonstrated in terms of a case study of a control system that has to adjust the height of a vehicle's suspension to different road conditions.

Keywords: software architectures, objects, collaborations, architectural styles, run-time adaptability, formal models.

1. Introduction

In the engineering of computer based systems, there has been a trend in which the quality of services delivered by a system, in terms of its dependability, performance and cost, is directly related to the quality and extent of the computer facilities embedded in that system. Software has played a central role in this trend because of its inherent flexibility in emulating physical devices and replacing human operators. As the life span of new emerging software intensive applications increases, so does the need for software to have the capability of adapting to changes that occur in its operating environment. However, providing an adaptive capability leads to an increase in software size and complexity, which could put system integrity at risk unless the software architecture enables adaptability to be engineered in a disciplined and structured manner.

Architectural structures for systems tend to abstract away from the details of a system, but assist in understanding broader system-level concerns. This can be achieved in software architectures by employing abstractions and notations which are appropriate for describing the software components, the interactions between these components, and the properties that regulate the composition of components. In architectural descriptions which use *collaboration-based designs* as a basis, software systems are represented as a composition of independently-definable collaborations: *collaborations* are a group of objects together with a group of activities that determine how objects interact, and the object's *role* is that part of an object which prescribes the activity of the object within a collaboration /Smaragdakis 98b/. However, there are some applications where the notion of collaboration is not sufficient to represent collaborative behaviour, for instance, in complex concurrent applications it is also

necessary to capture the notion of co-ordination for supporting error handling between multiple interacting objects /Randell 97, Xu 95/.

In addition to the notion of collaboration software architectures for adaptive systems should include an architectural element which is able to represent the dynamic composition of software components. The modelling abstraction *co-operative action* (CO action) was introduced in order to co-ordinate collaborations between objects, which can either be co-operative or competitive /de Lemos 98/. In a co-operative object-oriented architecture an object can be involved in more than one co-operation (defined by its participants and collaborative activity), and depending on the required behaviour of the system, co-operations are able to reconfigure interactions between objects.

The rest of the paper is organised as follows. Section 2 discusses some basic issues related with run-time adaptability, providing the motivation for the work. In section 3 we present a case study that will be used to illustrate the feasibility of representing adaptive software structures in terms of the co-operative object-oriented style. In section 4 the architectural style is defined in more detail by defining a meta-model for the notion of co-operative action. The architectural description of the case study is presented in section 5. In section 6, we present some related work in the area of collaboration-based designs, and finally, section 7 concludes with a discussion evaluating our contribution and indicating directions for future work.

2. Run-Time Adaptability

Run-time adaptability is the ability of a software system to adapt itself to changes that occur in its operating environment, while providing its required service. In a co-operative object-oriented architecture the degree of run-time adaptability of a software system depends on the flexibility of components changing their pattern of collaboration. Instead of having a software system based on components which individually are able to provide a wide range of services, the proposed approach relies on the ability of components to reconfigure their collaborations while they remain unchanged.

In the context of a co-operative object-oriented architecture, a system can either adapt its behaviour or its structure, although, adaptive systems might contain a mixture of these two types of adaptability.

- In behavioural adaptability, the system components and their configurations remain the same, while the system behaviour changes by modifying the pattern of collaboration between the components.
- In structural dependability, the behaviour of the system remains the same, while the system architecture changes by modifying (or replacing) the components, or their configurations. At the design level, an example of structural adaptability is adaptive fault tolerance /Kim 92/.

The aim of this paper is to define an architectural representation which enables behavioural adaptability to be incorporated in co-operative object-oriented architectures of software systems.

3. Description of the Case Study: Electronic Height Control System (EHCS)

The electronic height control system (EHCS) controls the height of a vehicle by regulating the individual heights of the wheels through a pneumatic suspension. The aim of this system

is to adjust the chassis level depending on the road conditions, in order to improve driving comfort and keep the headlight load-independent /Stauner 97/.

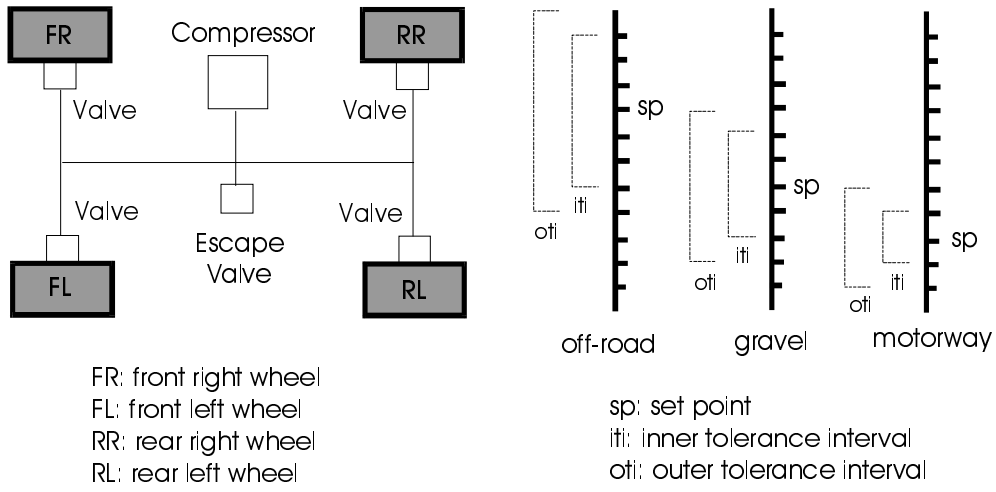


Figure 1. Diagramatic representation of the EHCS.

For this case study three distinct types of road are considered, namely, off-road, gravel and motorway. For each type of road we define a set point and two sets of tolerance intervals, as shown in the diagram of figure 1. In each of the four wheels there is an pneumatic suspension which is able to control the height of an individual wheel. Whenever the height of a wheel is outside the outer tolerance interval, the controller has to bring the height into the inner tolerance interval around the set point.

The major components of the EHCS are a valve and a height sensor at each wheel, and an escape valve and a compressor to be shared by all the wheels, as shown in the diagram of figure 1. The height of a wheel suspension is increased by opening the wheel valve, closing the escape valve, and pumping air into the suspension. The height is decreased by releasing air from the suspension by opening the escape valve, and the valve of the wheel from which the height has to be reduced. The compressor and the escape valve cannot be used simultaneously, priority is given to the compressor when both have to be used. It is assumed that the height values provided by the sensors are mean values of the actual readings from which the disturbances, like road holes, are eliminated.

The aim of this case study is to define a software architecture which enables the EHCS to adapt at run-time to changes that occur in the system environment. In terms of the height control system, the adaptability element is related with selection of the appropriate control algorithm depending on the type of road. In terms of the EHCS software architecture, the software components remain the same, while the pattern of collaboration between the components changes.

4. Co-operative Object-Oriented Style

An architectural style provides a specialised language for a specific class of systems that are related by shared structural and semantic properties /Shaw 96/, which include: a *vocabulary* of architectural elements (components and connectors), *configuration rules* that constraint how components and connectors can be composed, *semantic interpretations* that provide well-defined meanings for the components, connectors, and compositions of these, and the

type of *analyses* that can be performed on systems employing a particular style. For example, a software system might be described using one of the following more commonly used styles: pipes and filters, objects, repositories, layers, and interpreters.

Systems are defined by their components and the relationships among their components, hence when modelling systems using an object-oriented approach, objects alone are insufficient to describe the system behaviour. Co-operative Actions (CO actions) were introduced for representing interactions between objects which characterise collaborative behaviour /de Lemos 98/. One of the motivations for using CO actions in an object-oriented approach is the ability of CO actions to extract from the specification of an object those issues which are related with its collaborative activities (although preserving object's encapsulation property), thus avoiding a specification of a co-operation to be scattered among objects. CO actions are a variant of Co-ordinated Atomic Actions (CA actions) /Xu 95, Randell 97/ which are design mechanisms for structuring complex concurrent activities and supporting error recovery between multiple interacting objects in an object-oriented system. In the following, we present in more detail the co-operative object-oriented style, which adopts as a basis the features of object-oriented models.

4.1. Architectural Elements

The architectural elements of the co-operative object-oriented style are *classes* as the basic components, and *CO actions* as the basic connectors. (Classes and CO actions are instantiated, respectively, into objects and co-operations.) The difference between these elements is that while classes perform computation locally, CO actions essentially co-ordinate the distributed computation performed by the participant classes. In a CO action, the role of a class is prescribed by the activity of that class. A class may have as many roles as the number of CO actions it participates. The composition of these roles defines the interface of the class.

At the architectural level no relational information is spread across classes, only CO actions contain relational information (how a co-operative object-oriented architecture is implemented is discussed later, however a CO action can be instantiated into an association when the interactions between classes are simple service requests). An advantage for only CO actions to contain relational information is that, once a co-operative object-oriented architecture is instantiated, co-operations can be added or removed without interfering with the object implementation of objects, thus improving modularity and reusability of the software.

4.1.1. Classes

As in object-oriented models, classes in the proposed approach, support the representation of both structural and behavioural aspects of a system. A class is described by a template with the following fields: a **name**, declaration of **attributes** in terms of **constants** and **variables** which are local to the class, a description of its **structure** in terms of a collection of components in **composed of** and the **intra-relations** between the classes and its components, and finally, a description of the **behaviour** of the class. The behaviour field includes the **initial** state of the object, and behavioural **assumptions** or consistency invariants associated with the class. The behavioural field also includes the specification of the complete space of the behaviour of the class, in terms of its **normal**, **exceptional** and **failure** behaviours. Normal and exceptional behaviours are related with the liveness properties of a system

(“something good" eventually happens), while failure behaviours are related with the safety properties of a system (“something bad" does not happen).

4.1.2. Co-operative Actions

CO actions are employed in the specification of co-operative behaviour between classes. CO actions can either co-ordinate the activities to be performed by the classes, or execute some activity which is not associated with any particular class which takes part in the co-operation. A CO action is described by a template with the following fields: the CO action's **name**, declaration of **attributes** in terms of the names and types of the **participants** of the CO action, **constants** and **variables** local to the CO action, and the specification of the collaborative **behaviour** of the classes participating in the CO action. The template for describing a CO action is the following:

***CO Action*:**
attributes:
participants:
constants:
variables:
behaviour:
initial:
normal:
exceptional:
failure:

The **initial** state of a CO action represents its state when is activated, and is dissociated from the pre-conditions of the CO action: it either refers to the state of classes participating in the co-operation or the state of the variables local to the CO action. Associated with the description of **normal** behaviour, **pre-condition** and **post-condition** establish the respective conditions for a set of classes to start and finish a particular collaborative activity, and the **invariant** establishes the conditions which should hold while the collaborative activity is being performed. For the successful execution of a collaborative activity it is necessary that the pre- and post-conditions of the normal behaviour are satisfied, and that the invariant associated with the collaborative activity is not violated during its execution. For the specification of exceptional behaviour, the invariant is replaced by a **handler**, which identifies the exception event, together with the start and finish events associated with the handler of the exception. Although the pre-conditions for normal and exceptional behaviours are the same, the post-conditions for the exceptional behaviour might be different, depending on the degraded outcomes of a CO action, once an exception has occurred. In the definition of a CO action, an exception can be associated with the invariant whenever this is violated, or with the post-conditions whenever one of the conditions is not satisfied.

A CO action provides the basis for dealing with both co-operative and competitive concurrency by integrating two complementary concepts: *conversations* /Randell 75/ and *transactions* /Gray 93/. Conversational support is used to control co-operative concurrency and to implement co-ordinated and disciplined error recovery, whilst transactional support maintains the consistency of shared resources in the presence of failures and concurrency among different collaborative activities competing for these resources / Xu 95, Randell 97/.

4.2. Configuration Rules

For the description of systems, the configuration rules of the co-operative object-oriented style define how objects and co-operations can be combined. In the following, we will focus on the static properties of the co-operative object-oriented style.

In a co-operative object-oriented architecture each class and CO action has an unique name. Classes can participate in more than one CO action, and at least two classes have to be associated with a CO action, thus avoiding the "dangling" of CO actions. A CO action defines and is defined by the roles of the classes, thus creating the context in which classes collaborate. At the architectural level, the relationships between classes can be dependencies, generalisations, and aggregations. The same applies to CO actions.

4.3. Meta-Model of a Co-operative Action (CO Action)

In the following we define in more detail the concept of a CO action, according with the semantic description of UML /UML 97/. A CO action is considered as a specialisation of *Classifier* in the Core package of UML Foundation, which also includes the following specific forms: *Class*, *DataType* and *Interface*. The diagram of figure 2 shows the concrete constructs that define the structural backbone and the relationships of a CO action.

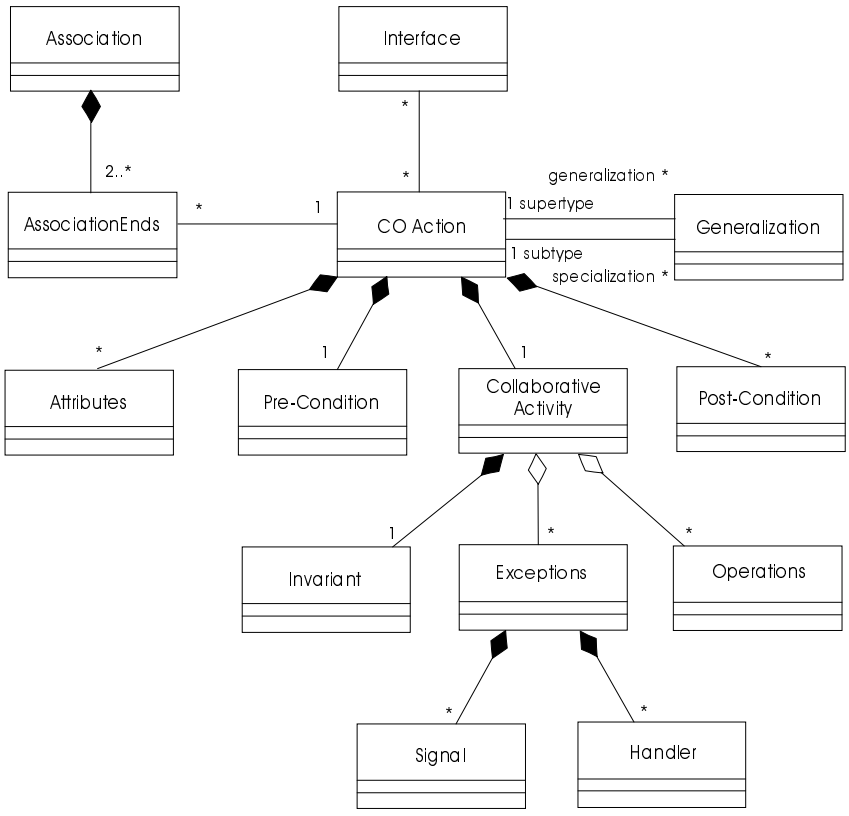


Figure 2. Meta-model of a CO action.

The purpose of a *CO action* is to declare the attributes and the collaborative activities that fully describe the structure and behaviour of co-operations. All the co-operations instantiated from a CO action will have attribute values matching the attributes of the CO action descriptor, and will support the collaborative activities defined by the CO action descriptor. An *Attribute* is a name property of a CO action that describes the range of values that instances of the property may hold. These attributes can either refer to remote attributes defined by the classes which take part in the co-operation, or local attributes to the CO action

which includes the list of participants that take part in the CO action. A *Collaborative Activity* is the implementation of a service that can effect the behaviour of two or more objects. Associated with the collaborative activity of a CO action there is a *Pre-condition* which defines the start of the activity, and one or more *Post-conditions* which define end of the activity. A collaborative activity of a CO action is specified in terms of a name, together with an *Invariant* which defines the collaborative activity, a set of *Operations* which establish the normal behaviour of the co-operation, and a set of *Exceptions* which establish the exceptional behaviour of the co-operation. The exceptions are defined in terms of exceptional *Events* and *Handlers*. The *Interface* of a CO action is the collection of collaborative activities which define the service to be delivered by the CO action.

An *Association* is a structural relationship that specifies a connection between classifiers, e.g. classes and CO actions. Associations are described in terms of a name, at least two *AssociationEnds* (which define the roles and the properties that should be observed of the classifier participating in the association), and a multiplicity property. An association may represent an aggregation between CO actions, but not between CO actions and classes. An aggregation specifies a whole-part relationship between the aggregate ("the whole") and a component ("the part"). Composition is a strong form of aggregation which requires that a part instance be included in at most one composite at time, although the owner may be changed over time.

A *Generalisation* is a taxonomic relationship between a more general element and a more specific element. A CO action can have generalisations to other CO actions, but not with classes. The full CO action descriptor of a CO action is derived by inheritance from its own segment declaration and those of its ancestors.

4.4. Co-operative Object-Oriented Architectures and Adaptability

In a co-operative object-oriented system, behavioural adaptability is obtained by changing how objects co-operate, and the selection of a co-operation depends on the state of the collaboration between the objects. An architectural representation of such system should describe the collaborative activities between classes in terms of CO actions. The conditions for selecting a co-operation should be part of the definition of a CO action, and these conditions are related to either the internal state of the co-operation or the states of the objects participating in the co-operation. Hence the architectural representation should be able to describe, across different states, the behavioural adaptability of the collaborative activities between classes.

The intent of the *State* design pattern is to allow an object to alter its behaviour when its internal structure changes /Gamma 94/. In this paper we claim that this design pattern can be also used to provide the required support for a co-operation to alter its behaviour when its state changes. The structure of the design pattern *State*, in terms of CO actions, is shown in figure 2. The abstract *State* CO action defines the interface common to all the CO actions that represent the different states of the co-operation (an instance of COAction). COAction delegates all state-specific collaborations to the *State* CO action, and depending on its state, COAction uses an instance of a specialised CO actions (COAction1, COAction2,...) of the abstract *State* CO action to implement a collaboration. Using the design pattern *State*, we are able to obtain an effective and structured representation of behavioural adaptability using the co-operative object-oriented style.

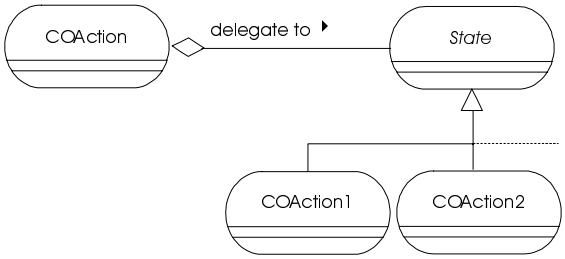


Figure 3. *State* design pattern in terms of CO actions.

4.5. General Features of a Co-operative Object-Oriented Architectures

The general features of a co-operative object-oriented architecture are presented in the following, together with some examples taken from the case study:

- Design-time *evolution* is achieved from the capability that co-operations have for implementing changes that otherwise should be incorporated into objects /Tramontana 99a, Tramontana 99b/: libraries contain fewer and lower complexity components, while special features would be provided by the co-operations.

  For example, the impact of replacing the compressor assemble, which might contain a physical and a software component, will be minimal for the rest of the system because the changes in the software would be restricted to the CO actions in which the compressor is a participant. Also, if other types of road were considered, the changes in the software would be restricted of adding new CO actions which incorporate the control algorithms which are appropriate for the new type of roads.

- Run-time *adaptability* is achieved from the flexibility that co-operations have for enabling objects to change their pattern of collaboration: objects are rigid entities, how they collaborate provide the basis for adaptability.

  For example, if the pneumatic suspension of a wheel fails, the CO action which co-ordinates the control of the chassis level can have alternative collaborations to compensate the presence of a faulty suspension, by regulating the height of the remaining three wheels.

- *Fault tolerance* is achieved by using co-operations as an error containment mechanism /de Lemos 99b/: the role of co-operations is to co-ordinate the handling of exceptions between collaborating objects.

  For example, if one of the valves of the wheels fail (either stuck open or close), the CO actions local to that wheel will attempt to recover from the temporary fault. However, if the recovery fails, the CO action should then signal an exception to a higher level CO action which is responsible for adjusting, in a co-ordinated manner, the heights of the other wheels.

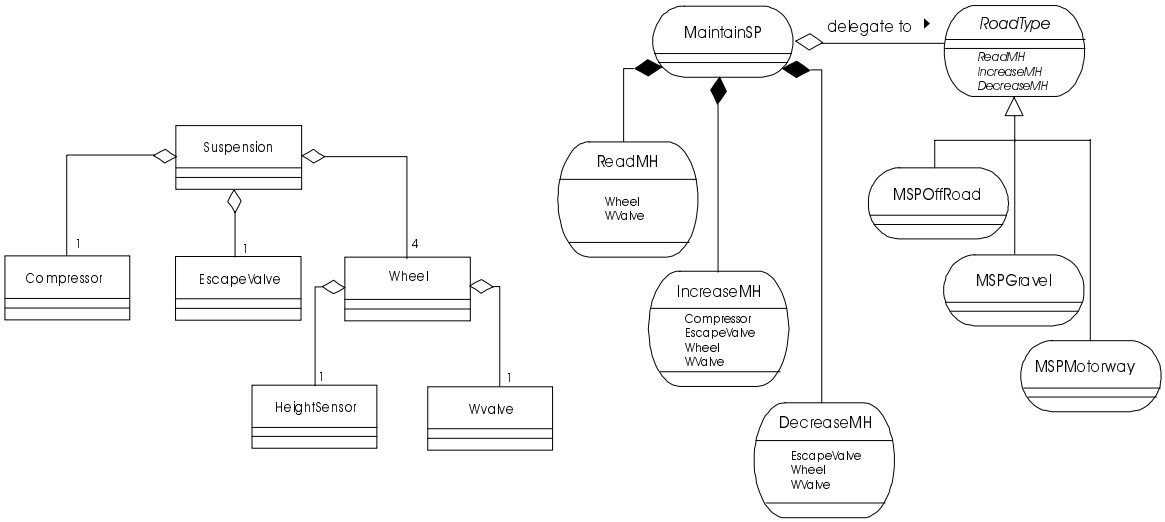
- *Safety analysis* is performed more effectively by extracting from the objects the behavioural dependencies associated with their interactions /de Lemos 99c/: aggregate behaviours of objects can be modelled and analysed without the need for modelling the entire system.

For example, when analysing interactions between the controllers of the individual wheels, there is no need to model and analyse the behaviour of the entire EHCS, instead the safety analysis can focus on the CO actions which capture the roles of the classes which are involved in the interactions.

5. A Software Architecture for the EHCS

In this section the software architecture for the electronic height control system (EHCS) of a vehicle suspension is established in terms of the co-operative object-oriented style, previously defined. The diagram of figure 4 represents the class and CO action structures for the EHCS. The representation of CO actions follows that of a class in UML /Booch 98/, except for the rounded corners.

The CO action MaintainSP is responsible for maintaining the height of the suspension around the set point, and is composed by three other CO actions: ReadMH which is responsible for updating Wheel with the value of mean height of the suspension, and IncreaseMH and DecreaseMH which are responsible, respectively, for increasing and decreasing the suspension height of a wheel. Depending on road condition, a different control algorithm is necessary for maintaining the height of the suspension, hence MaintainSP can be specialised into MSPOffRoad, MSPGravel, and MSPMotorway. For example, it might be necessary, depending on the road conditions, to establish different tolerance levels for IncreaseMH and DecreaseMH, or to establish time intervals for updating (Δupdate) the value of the mean height in class Wheel.



(i) class diagram (ii) CO action diagram

Figure 4. Co-operative object-oriented diagrams for the EHCS.

In this paper we have chosen to specify the dynamic behaviour of CO actions using a property oriented formalism instead of operational formalism (like Statecharts which is part of UML) because the purpose of this paper is to focus on the properties of an adaptive system, rather than on how a design should be implemented. In the following, the CO actions will be formally specified in terms of Extended Real-Time Logic (ERTL) /de Lemos 96, Hall 96/ (an outline of ERTL is presented in the Appendix) following the template previously presented. In this paper, the behavioural specification of CO actions will be restricted to the

normal behaviour. It is not in the scope of this paper to deal with exceptional and failure behaviours, which were presented elsewhere /de Lemos 99a, de Lemos 99b/. For the sake of brevity, and to avoid repetition, only the composite CO action MaintainSP will be specified. The specifications of MSPOffRoad, MSPGravel, and MSPMotorway follow directly from MaintainSP. We assume that MaintainSP, which is considered appropriate for all types of road, is replaced by other three CO actions which encapsulate control algorithms which are specific for particular types of road.

The CO action MaintainSP co-ordinates the activities between the components of the Suspension to maintain the mean height of a Wheel's around the established set point. The co-ordination of the collaborative activities is partitioned into three CO actions, detailed below. The definition of MaintainSP states that the pre-condition for MaintainSP to be activated is when EHCS is switched on (ehcs.on), and it will remain activated until the EHCS is switched off. The invariant defines the type of road for which MaintainSP is appropriate, which for this case we assume to be for all types of road. The provision of an adaptive software will be based on the specialisation of MaintainSP for the different types of road, which will be presented at the end of this section.

MaintainSP:

attributes:

participants:

c	*Compressor*
ev	*Valve*
w	*Wheel*
w.hs	*HeightSensor*
w.wv	*Valve*
ehcs	*EHCS*

variables:

behaviour:

initial:

$\Phi(\neg ehcs.on, 1, 0)$

normal:

pre-condition:

$\forall t \bullet \forall i \in \Im^{+}: \Theta(\nearrow maintainSP, i, t) \Leftrightarrow \Theta(\nearrow ehcs.on, i, t)$

invariant:

$\forall t \bullet \forall i \in \Im^{+}: \Phi(maintainSP, i, t) \Leftrightarrow \Phi(w.tr=all, i, t)$

operations:

ReadMH
IncreaseMH
DecreaseMH

post-condition:

$\forall t \bullet \forall i \in \Im^{+}: \Theta(\searrow maintainSP, i, t) \Leftrightarrow \Theta(\searrow ehcs.on, i, t)$

The CO action ReadMH captures the collaboration between Wheel and HeightSensor, and is responsible for updating periodically the Wheel's variable for the mean height of the suspension, which is obtained from the HeightSensor. The pre-condition for normal behaviour establishes that ReadMH starts periodically every Δupdate. The invariant states that for ReadMH to be active, the current update has still to be made and the interval for the next reading has not expired. The post-condition is captured by two transition event predicates which specify the necessary and sufficient conditions for the co-operation to end: the variable w.height has been updated, or the time interval available for updating has expired.

ReadMH:

attributes:

participants:

w	*Wheel*
w.hs	*HeightSensor*

variables:

Δupdate	*Real*

behaviour:

initial:

normal:

pre-condition:

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\nearrow \text{readMH}, i, t) \Leftrightarrow \exists t_1 \bullet \Theta(\nearrow \text{readMH}, i, t_1) \wedge t \geq t_1 - \Delta\text{update}$

invariant:

$\forall t \bullet \forall i \in \Im^+$: $\Phi(\text{readMH}, i, t) \Leftrightarrow$
$\Phi(\text{w.height} \neq \text{w.hs.value}, i, t) \wedge (\exists t_1 \bullet \Theta(\nearrow \text{readMH}, i, t_1) \wedge t < t_1 + \Delta\text{update})$

post-condition:

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\searrow \text{readMH}, i, t) \Leftrightarrow$
$\Theta(\nearrow(\text{w.height} = \text{w.hs.value}), i, t) \vee (\exists t_1 \bullet \Theta(\nearrow \text{readMH}, i, t_1) \wedge t \geq t_1 - \Delta\text{update})$

The CO action IncreaseMH is responsible for increasing the mean height of the suspension once a minimum threshold is reached. The pre-condition for normal behaviour establishes that the CO action starts when the Compressor is off, the EscapeValve and WheelValve are closed, and the minimum height threshold is reached. While the mean height of the suspension is being increased the Compressor should be on, the EscapeValve closed, the WheelValve open. Once the mean height is within the inner tolerance interval, IncreaseMH ceases to be active.

IncreaseMH:

attributes:

participants:

c	*Compressor*
ev	*Valve*
w	*Wheel*
w.hs	*HeightSensor*
w.wv	*Valve*

variables:

iti, oti	*Real*

behaviour:

initial:

$\Phi(\neg \text{c.on} \wedge \neg \text{ev.open} \wedge \neg \text{w.wv.open}, 1, 0)$

normal:

pre-condition:

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\nearrow \text{increaseMH}, i, t) \Leftrightarrow$
$\Theta(\nearrow(\neg \text{c.on} \wedge \neg \text{ev.open} \wedge \neg\ \text{w.wv.open} \wedge (\text{w.height} < (\text{w.setPoint} - \text{oti}/2))), i, t)$

invariant:

$\forall t \bullet \forall i \in \Im^+$: $\Phi(\text{increaseMH}, i, t) \Leftrightarrow \Phi(\text{c.on} \wedge \neg \text{ev.open} \wedge \text{w.wv.open}, i, t)$

post-condition:

$\forall t \bullet \forall i \in \Im^+$: $\Theta(\searrow \text{increaseMH}, i, t) \Leftrightarrow$
$\Theta(\nearrow(\neg \text{c.on} \wedge \neg \text{ev.open} \wedge \neg\ \text{w.wv.open} \wedge$
$((\text{w.setPoint} + \text{iti}/2) > \text{w.height} > (\text{w.setPoint} - \text{iti}/2))), i, t)$

The description of CO action DecreaseMH follows the same pattern of IncreaseMH. However, for reducing the mean height of the suspension the Compressor should be off, and the EscapeValve and WheelValve should be open.

DecreaseMH:

attributes:

participants:

c	*Compressor*
ev	*Valve*
w	*Wheel*
w.hs	*HeightSensor*
w.wv	*Valve*

variables:

iti, oti	*Real*

behaviour:

initial:

$\Phi(\neg c.on \wedge \neg ev.open \wedge \neg\ w.wv.open, 1, 0)$

normal:

pre-condition:

$\forall t \bullet \forall i \in \Im^{+}: \Theta(\nearrow decreaseMH, i, t) \Leftrightarrow$

$\Theta(\nearrow(\neg c.on \wedge \neg ev.open \wedge \neg\ w.wv.open \wedge (w.height > (w.setPoint + oti/2))), i, t)$

invariant:

$\forall t \bullet \forall i \in \Im^{+}: \Phi(decreaseMH, i, t) \Leftrightarrow \Phi(\neg c.on \wedge ev.open \wedge w.wv.open, i, t)$

post-condition:

$\forall t \bullet \forall i \in \Im^{+}: \Theta(\searrow decreaseMH, i, t) \Leftrightarrow$

$\Theta(\nearrow(\neg c.on \wedge \neg ev.open \wedge \neg\ w.wv.open \wedge$
$((w.setPoint + iti/2) > w.height > (w.setPoint - iti/2))), i, t)$

According with the proposed approach, outlined in section 4.4, an adaptable software system for the EHCS can be obtained by applying the design pattern *State* /Gamma 94/ to the CO actions that capture the collaborative activity between the components of the Suspension. Referring to the CO action diagram of figure 4, the behaviour of MaintainSP depends on the state of *RoadType*, and depending on this state the behaviour of MaintainSP must change at run-time. Instead of defining a CO action for all types of road, the design pattern *State* allows to partition MaintainSP into other CO actions, namely, MSPOffRoad, MSPGravel and MSPMotorway. These three CO actions are mutually independent, and depending on the types of road they are defined in terms of specific attributes and control algorithms for maintaining the height of the suspension around the established set point, as shown in figure 1.

6. Related Work

Most of the work in the definition of architectures for adaptive systems has focused on the architecture of the entire system by defining the roles of specific components, like planners, schedulers, and interfaces /Muller 91/, rather then focusing on the definition of modelling abstractions which can be used for structuring systems and applications. The work described in this paper investigates how the co-operative object-oriented style can be used in establishing software architectures for adaptive systems. The proposed architectural has similar features to collaboration-based designs which deal mostly with the design-time adaptability, rather than run-time adaptability.

The notions behind collaboration-based designs, which aim to explicitly specify the interrelations between objects in an object-oriented model, are not new. Although differences might exist between the existing approaches for representing these abstractions, in general terms, collaborations are known as a group of objects together with a group of activities that determine how objects interact. In a collaboration-based design the aim is to compose independently-definable collaborations when defining software systems. Some of the collaboration-based approaches have adopted the restricted view that collaborations should be used to model message passing and state changes, by focusing on the representation of roles that an object has while participating in a collaboration /Smaragdakis 98a, VanHilst 96/ (although in /Smaragdakis 98b/ the definition of *collaboration-components* is based on the roles of a collaboration).

On the other hand, similar to the co-operative object-oriented style, there are those approaches which have adopted a broader view in which collaborations are modelling abstraction which are able to capture the properties of a collaborative activity to be performed by a group of objects /Helm 90, Kristensen 96, Kurki-Suonio 96/. *Contracts* were introduced in /Helm 90/ as abstractions to specify behavioural compositions and obligations on participating objects. A contract defines a set of communicating participants and their contractual obligations, pre-conditions which are required for participants to establish a contract, and an invariant which has to be maintained by these participants. *Activities* were introduced in /Kristensen 96/ as an abstraction mechanisms to model the interplay between objects, and it is claimed that activities are more powerful than the *Mediator* pattern /Gamma 94/ because they can be used as building mechanisms in creating modelling abstractions which can then represent concepts, such as roles and relations. The Catalysis approach defines a design notation which employs the notions of *joint action* and *collaboration* to represent, respectively, activities and participants /D'Souza 98/, instead of having an unique recursive notion like co-operative actions (CO actions). The use of *connectors* to co-ordinate the activity between objects by intercepting messages to decide the invocation of a participant /Ducasse 98/ is another approach that has similar intentions to that of CO actions, although its representation capabilities are more restricted. The theoretical foundation of the incremental derivation of collective behaviour of operational models of objects was defined in /Kurki-Suonio 96/ in the context of an action-oriented language, however issues like nested actions are not considered.

One issue outside the scope of this paper that has not been mentioned, concerns the implementation of collaboration-based designs. The roles in collaborations can be implemented using standard C++ class templates by binding its parameters to specific classes representing the participants /VanHilst 96/. Collaborations can be implemented through inheritance of nested classes using standard C++ class templates parameterized by the roles it uses /Smaragdakis 98b/. A collaboration can be described by specifying a set of roles and their correspondent activity, in which activities and roles are implemented, respectively, as relation-classes and role-classes, both non-standard C++ classes /Kristensen 96/. The co-operative object-oriented architecture, proposed in this paper, has been implemented using the *Mediator* /Gamma 94/ design pattern /de Lemos 99d/, and currently we are investigating the use of reflection for implementing CO actions as meta-objects /Tramontana 99a/.

7. Conclusion

In this paper we have presented how collaborations between objects can be exploited when defining software architectures for adaptive systems. As a basis for the proposed approach, we have considered that objects are rigid entities, and the basis for adaptability depends on

how they collaborate. Hence all the additional features which enables an object, or a group of objects, to adapt to changes that occur in its environment are not capture in the object itself, instead they are defined in the co-operations in which the object is a participant. For describing the architectures for adaptive software systems we have defined a co-operative object-oriented style where components are the classes, and connectors are the co-operative actions (CO actions). As architectural elements, CO actions capture the behavioural dependencies between the classes which are related with the adaptability features of an object, or group of objects.

Although the definition of a CO action was presented in the context of components (objects) which have very simple structures, the aim of the work is to obtain a more general definition of a CO action which can be used as a sophisticated connector for structurally more complex software components. For that, it might be necessary to define a CO action as an architectural pattern (or framework) which can be instantiated into several domain related applications. Also in this paper, we have only considered the type of run-time adaptability where the components remains unchanged while the behaviour of the system changes, however, depending on the type of application and the purpose of the system, another types of adaptability have also to be considered.

Acknowledgements

I would like to thank Alexander Romanovsky and Emiliano Tramontana for the discussions on the topic, and to acknowledge the financial support from the EPSRC/UK ADAPT Project.

References

/Booch 98/ G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley. Reading, MA. 1998.

/de Lemos 96/ R. de Lemos, J. G. Hall. “Extended RTL in the Specification and Verification of an Industrial Press”. *Hybrid Systems III*. Lecture Notes in Computer Science 1066. Eds. R. Alur, T. A. Henzinger, E. Sontag. Springer-Verlag. Berlin, Germany. 1996. pp. 114-125.

/de Lemos 98/ R. de Lemos, A. Romanovsky. “Coordinated Atomic Actions in Modelling Object Cooperation” *Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Kyoto, Japan. April, 1998. pp. 152-161.

/de Lemos 99a/ R. de Lemos, A. Saeed. *Validating Formal Verification using Safety Analysis Techniques*. Technical Report 668. Department of Computing Science. University of Newcastle upon Tyne. UK. 1999.

/de Lemos 99b/ R. de Lemos, A. Romanovsky. “Exception Handling in a Cooperative Object-Oriented Approach”. *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Saint Malo, France. May, 1999. (to appear)

/de Lemos 99c/ R. de Lemos. “Analysis of the Safety Properties of a System from the Viewpoint of its Components Interactions”. *9th Brazilian Conference on Fault Tolerant Systems*. Campinas, SP. Brazil. July 1999. (to appear)

/de Lemos 99d/ R. de Lemos. *A Co-operative Object-Oriented Style for Control Systems*. Technical Report. Department of Computing Science. University of Newcastle upon Tyne. UK. 1999.

/D'Souza 98/ D. F. D'Souza, A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley. Reading, MA. 1998.

/Ducasse 98/ S. Ducasse, M. Günter. "Coordination of Active Objects by Means of Explict Connectors". *Coordination Technologies for Information Systems (CTIS'98)*. Vienna, Austria. August, 1998.

/Gamma 94/ E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. Reading, MA. 1994.

/Gray 93/ J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Kaufman Publishers. San Mateo, CA. 1993.

/Hall 96/ J. G. Hall, R. de Lemos. "ERTL: an Extension to RTL for the Specification, Analysis and Verification of Hybrid Systems". *Proceedings of the 8th EUROMICRO Workshop on Real-Time Systems 96*. L'Aquila, Italy. June 1996. pp. 3-8.

/Helm 90/ R. Helm, I. M. Holland, D. Ganopadhyay. "Contracts: Specifying Behavioural Compositions in Object-Oriented Systems". *Proceeedings of ECOOP/OOPSLA'90, SIGPLAN Notices 25(10)*. October 1990. pp. 169-180.

/Jahanian 86/ F. Jahanian, A. Mok. "Safety Analysis of Timing Properties in Real-Time Systems". *IEEE Transactions on Software Engineering Vol. SE-12(9)*. September 1986. pp. 890-904.

/Jahanian 88/ F. Jahanian, A. Mok, D. A. Stuart. *Formal Specifications of Real-Time Systems*. Technical Report TR-88-25, Department of Computer Science, University of Texas at Austin, TX. June 1988.

/Kim 92/ K. H. Kim, T. Lawrence. "Adaptive Fault Tolerance in Complex Real-Time Distributed Applications". *Computer Communication, Vol. 15(4)*. May 1992. pp. 243-251.

/Kristensen 96/ B. B. Kristensen, and D. C. M. May. "Activities: Abstractions for Collective Behaviour". *Proceedings of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*. Lecture Notes in Computer Science 1098. Ed. P. Cointe. Linz, Austria. July 1996. pp. 472-501.

/Kurki-Suonio 96/ R. Kurki-Suonio. "Fundamentals of Object-Oriented Specification and Modelling of Collective Behaviours". *Object-Oriented Behavioural Specifications*. Eds. H. Kilov, and W. Harvey. Kluwer Academic Publishers. Boston, MA. 1996. pp. 101-119.

/Muller/ J. P. Muller. *The Design of Intelligent Agents: A Layered Approach*. Springer-Verlag. Berlin, Germany. 1991

/Randell 75/ B. Randell. "System Structure for Software Fault-Tolerance". *IEEE Transactions on Software Engineering Vol. SE -1(2)*. 1975. pp. 220-232.

/Randell 97/ B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, A. F. Zorzo. *Co-ordinated Atomic Actions: from Concept to Implementation*. Technical Report 595. Department of Computing Science. University of Newcastle. UK. 1997.

/Shaw 96/ M. Shaw, D. Garlan. *Software Architectures: Perspectives on an Emerging Discipline*. Prentice-Hall, Inc. Upper Saddle River, NJ. 1996.

/Smaragdakis 98a/ Y. Smaragdakis, D. Batory. "Implementing Layered Designs with Mixin Layers". *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*. Lecture Notes in Computer Science 1445. Ed. E. Jul. Brussels, Belgium. July 1998. pp. 550-570.

/Smaragdakis 98b/ Y. Smaragdakis, D. Batory. "Implementing Reusable Object-Oriented". *Proceedings of the 5th International Conference on Software Reuse (ICSR'98)*. Victoria, Canada. June 1998.

/Stauner 97/ T. Stauner, O. Müller, M. Fuchs. "Using HYTECH to Verify an Automative Control System". *Hybrid and Real-Time Systems*. Ed. O. Maler. Lecture Notes in Computer Science 1201. Springer-Verlag. Berlin, Germany. 1997. pp. 139-153.

/Tramontana 99a/ E. Tramontana, R. de Lemos. *A Reflective Approach for Describing Co-operation between Objects*. Technical Report. Department of Computing Science. University of Newcastle upon Tyne. UK. 1999.

/Tramontana 99b/ E. Tramontana, R. de Lemos. "Design and Implementation of Evolvable Software using Reflection". *Workshop on Software Change and Evolution (SCE'99)*. Los Angeles, CA. May, 1999. (accepted)

/UML 97/ UML Semantics (version 1.1). Rational Software. September 1997.

/VanHilst 96/ M. VanHilst, D. Notkin. "Using Role Components to Implement Collaboration-Based Designs". *Proceedings of the 11th Annual ACM Conference on Object-Oriented, Programming Systems, Languages and Applications (OOPSLA'96)*. San Jose, CA. October 1996. pp. 359-369.

/Xu 95/ J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z. Wu. "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery". *Proceedings of the 25th International. Symposium on Fault-Tolerant Computing*. Pasadena, CA. 1995. pp. 499-509.

/Xu 98/ J. Xu, A. Romanovsky, B. Randell, "Co-ordinated Exception Handling in Distributed Object Systems: from Model to System Implementation". *Proceedings of the 18th International Conference on Distributed Computing Systems*. IEEE CS Press. Amsterdam, The Netherlands. May 1998. pp. 12-21.

Appendix - Extended Real Time Logic (ERTL)

The behavioural specification of both classes and CO actions will be made in terms of the event-action model which provides a set of primitive concepts for the modelling and analysis of phenomena associated with the computer system and its environment. In the event-action model, an *event* serves as a temporal marker, an *action* is an operation which consumes a

bounded quantity of resources, and a *system predicate* is an assertion about a system variable at a time point.

Extended Real Time Logic (ERTL) /de Lemos 96, Hall 96/ is a first order predicate logic for the modelling and analysis of hybrid systems, taking as a basis Jahanian & Mok's Real Time Logic (RTL) /Jahanian 86, Jahanian 88/. RTL uses uninterpreted predicates to relate events of a system to the time of their occurrence, thereby providing the means for reasoning about absolute timing properties of real-time systems. The extensions provided by ERTL allow reasoning about system behaviour in both value and time domains through predicates defined in terms of system variables.

The *occurrence relation* (Θ) captures the notion of real time by assigning a time value to each occurrence of an event. $\Theta(e, i, t)$ defines that the ith occurrence of event e occurs at time t.

$\forall t \bullet \forall i \in P: \Theta(\text{Motor\_On}, i, t)$

The ith occurrence of event MotorOn has occurred at time t.

A *transition event* is defined by the transition of a system predicate from false to true, or from true to false, at a particular time point. For a system predicate P, the respective transition events are $\nearrow P$ and $\searrow P$.

$\forall t \bullet \forall i \in P: \Theta(\searrow \text{plateOnBeg}, i, t) \Leftrightarrow \Theta(\nearrow(\text{plateOnEnd} \wedge \neg \text{beltOn}), i, t)$

The transition event which captures the instant which of the predicate plateOnBeg becomes false is equivalent to the transition event which captures the instant that the conjunction of plateOnBeg and the negation of beltOn becomes true.

The *holding relation* (Φ) captures whether a system predicate holds true at a time point. $\Phi(f, i, t)$ defines that a formula f holds for the ith time, at time t.

$\forall t \bullet \forall i \in P: \Phi(\text{moveDown}, i, t) \Leftrightarrow \Phi(\neg \text{bottom} \wedge \neg \text{plateOn}, i, t)$

The predicate moveDown holds true *iff* the conjunction of the negating predicates bottom and plateOn also holds true.